AMENDMENTS

IN THE SPECIFICATION:

Please replace the title with the following amended title.

METHOD AND APPARATUS FOR CONTROLLING EXECUTION OF A CHILD PROCESS GENERATED BY A MODIFIED PARENT PROCESS ~~BEING MONITORED UNDER AN INSTRUMENTATION SCHEME~~

Please replace paragraph [0001] with the following amended paragraph.

[0006] The computing community has developed tools and methods to analyze the run-time behavior of a computer program. Many of the tools and methods use statistical sampling and binary instrumentation techniques. Statistical sampling is performed by recording periodic snapshots of the program's state, *e.g.*, the program's instruction pointer. Sampling imposes a low overhead on a program's run time performance, is relatively non-intrusive, and imprecise. For example, it may be difficult to associate a sampled instruction pointer ~~may not be related to~~ with the particular instruction ~~address~~ that caused ~~a particular~~ the latest sampling event.

Please replace paragraph [0002] with the following amended paragraph.

[0007] While binary instrumentation leads to more precise results, the accuracy comes at some cost to the run-time performance of the instrumented program. Traditional binary instrumentation is static. It involves rewriting the whole program before any run to insert data-gathering code. Because the binary code of a program is modified, all interactions with the processor and the operating system can change significantly. ~~For example, additional instructions and changes to a program's cache and paging behaviors can cause run-time performance increases from a few percent up to 400%.~~ Consequently, binary instrumentation is considered intrusive.

Please replace paragraph [0003] with the following amended paragraph.

[0008] Dynamic binary instrumentation allows program instructions to be changed on-the-fly and leads to a whole class of more precise run-time monitoring results. Unlike static binary instrumentation techniques that are applied over an entire program prior to execution of the program,[.] Dynamic ~~dynamic~~ binary instrumentation is performed at run-time of a program and only instruments those portions of an executable that are executed. Consequently, dynamic binary instrumentation techniques can significantly reduce the overhead imposed by the instrumentation process.

Please replace paragraph [0005] with the following amended paragraph.

[0005] A basic reason for the difficulty in testing the correctness and performance of a program is that program behavior largely depends on the data on which the program operates and, in the case of interactive programs, on the information (data and commands) received from a user. Therefore, even if exhaustive testing is impossible, as is often the case, program testing and performance analysis ~~verification~~ is preferably conducted by causing the program to operate with some data. The act of executing a program entails the creation of one or multiple "processes." ~~In other words, by creating and performing what is defined as a "process" by software designers.~~

Please replace paragraph [0006] with the following amended paragraph.

[0006] A "process" is commonly defined as an address space, one or multiple ~~a~~ control threads operating within the address space, and the set of system resources needed for operating with the threads. Therefore, a "process" is a logic entity consisting of the program itself, the data on which it must operate, the memory resources, and input / output resources. Executing a given program may lead to multiple processes being created. Program verification and performance testing encompasses execution of the program ~~as a process thread~~ to test if the process or

processes develop ~~develops~~ in the correct way or if undesired or unexpected events occur.

Please replace paragraph [0007] with the following amended paragraph.

[0007] Generally, software development tools use two basic techniques to monitor the execution of a given process, in-process monitoring and out-of-process monitoring. In-process monitoring involves modifying a program to be tested so that select program instructions are preceded and followed by overhead instructions that extract variable information, control execution of the instruction, and monitor program execution. With out-of-process monitoring, a monitoring program executes in a different process and interacts with the monitored processes. ~~Controllably execute program instructions, tracing functions, or tracers and symbolic analysis functions, or symbolic debuggers.~~

Please replace paragraph [0008] with the following amended paragraph.

[0008] Symbolic debuggers constitute an example of out-of-process monitoring. Symbolic debuggers are interactive programs which allow a software engineer to monitor the execution of a compiled program. The user can follow the execution of a compiled program in a familiar high-level programming language while the program to be tested executes. ~~Tracing functions modify a program to be tested so that select program instructions are preceded and followed by overhead instructions that extract variable information, control execution of the instruction, and can monitor program execution. Symbolic debuggers are interactive programs, which translate a high-level language source program to be tested into a compiled program.~~ Symbolic debuggers modify an executable copy of the source by selectively inserting conditional branches to other routines, instruction sequences, and break points. The compiled and instrumented program can then be run under the control of a managing program or a software engineer via a human-machine interface.

Please replace paragraph [0009] with the following amended paragraph.

[0009] Symbolic debuggers also enable the insertion of instruction sequences for recording variables used in execution of the instruction and, on user request, can add and remove break points, modify variables, and permit modification of the hardware environment. These techniques are particularly effective in that they permit step-by-step control of the execution of a program, that is, they allow the evolution of the related process to be controlled by halting and restarting the process at will and by changing parameters during the course of execution of the process. The tools also can display the execution status of the process to the software engineer in detail by means of display windows or other output devices that enable the user to continuously monitor the program. Some tools automate the process of setting break points in the executable version of the source code.

Please replace paragraph [0012] with the following amended paragraph.

[0012] Moreover, software development tools that use symbolic debuggers can encounter deadlock conditions that result from the standard execution of operating system level instructions. ~~One operating system that has gained widespread acceptance is the UNIX® operating system. UNIX® is a trademark of the American Telephone and Telegraph Company of New York, New York, U.S.A.~~

Please replace paragraph [0013] with the following amended paragraph.

[0013] One operating system that has gained widespread acceptance is the UNIX® operating system. UNIX® is a trademark of the American Telephone and Telegraph Company of New York, New York, U.S.A. The UNIX® operating system is a multi-user, time-sharing operating system with a tree-structured file system. Other noteworthy functional features are its logical I/O capabilities, pipes, and forks. The logical I/O capabilities allow a user to specify the input and output files of a program at runtime rather than at compile time, thus providing greater flexibility. Piping is a

feature that enables buffering of input and output data to and from other processes. Forking is a feature that enables the creation of a new process.

Please replace paragraph [0015] with the following amended paragraph.

[0015] The popularity of the UNIX® operating system has led to the creation of numerous open source and proprietary variations such as LINUX®, HP-UX®, ~~PRIMIX®~~, *etc.* LINUX® is a trademark of William R. Della-Croce, Jr. (individual) of Boston, Massachusetts, U.S.A.. HP-UX®, is a trademark of the Hewlett-Packard Company, of Palo Alto, California, U.S.A. ~~PRIMIX® is a trademark of Primix Solutions, Inc. of Watertown, Massachusetts, U.S.A. These~~ variants Variants of the UNIX® operating system inherently use the UNIX® operating system's logical I/O capabilities, pipes, and forks.

Please replace paragraph [0016] with the following amended paragraph.

[0016] Software development tools can encounter a deadlock condition when a process under test includes a "vfork" system call ~~"fork" instruction~~. The operation of a "vfork" system call ~~"fork" instruction~~ in the UNIX® operating system involves spawning a new process and copying the process image of the parent (the process making the vfork ~~fork~~ call) to the child process (the newly spawned process).

Please insert paragraphs [0017] through [0021].

[0017] Monitoring facilities enable a process or thread to control the execution of threads running in another process. Generally, monitoring facilities control other threads by reading and modifying the state of the process. "Thread trace," also known as "ttrace" is a tracing facility for single and multithreaded processes. Ttrace is an evolution of "process trace," also known as "ptrace." A monitoring facility typically allows the monitoring program to declare an interest in the occurrence of particular events associated with any thread or for a specific subset of threads. For example, the monitoring process may want to be informed when a thread receives a signal, invokes a system call, or executes a breakpoint. While under the control of a monitoring

facility, the monitored code behaves normally until one of those events occur. At this point, the thread or process enters a stopped or suspended state and the tracing process is notified of the event. In the ttrace facility, the monitoring process receives such notifications by invoking the system call ttrace_wait.

[0018] Ttrace_wait can be called in blocking or non-blocking modes. When called in blocking mode, the system call will not return until an event is available. In non-blocking mode the system call will return promptly but may indicate that no event notification is available. When event notification is available, ttrace_wait will provide an indication of the event type encountered.

[0019] At a given point in time, a monitoring program may monitor multiple processes, each process including one or multiple threads. Monitoring facilities such as ttrace typically provide a way to separate event notifications from the various processes, or even from the various threads in a given process. In the case of ttrace_wait, the monitoring process will either return events from any process monitored, from any thread in a specified process, or from a specified thread in a specified process. Using the mode where a single ttrace_wait call will provide information about any process monitored can introduce a bottleneck or make run-time analysis too complex.

[0020] A vfork system call differs from a fork system call in that the child process created via the vfork system call shares the same address space as the parent until exec or _exit is called, while a child process created via the fork system call gets a copy of the parent address space. In some operating systems, the vfork system call is implemented by having the parent process blocked until the child process calls exec or _exit. This allows for a simple implementation of vfork, as the child process can be created cheaply as it directly uses most of the structures associated with the parent process.

[0021] As mentioned above, such an implementation of the vfork system call can present a deadlock condition for debuggers that use known tracing facilities. If the monitoring process is using monitoring facilities that in turn are using per-process notifications, the monitoring process will not be aware of the newly created child process, until the monitoring process is notified that the child process exists. In some existing designs, however, notification is not delivered until the parent process is unblocked. That is, when the child process calls exec or _exit. Furthermore, the child

process itself can be blocked as it generates monitoring events. This results in a deadlock, as both the traced parent and child process are blocked while the tracing process waits for an indication of an event that cannot be delivered until the child process is unblocked.

Please replace paragraph [0022] (original paragraph [0017]) with the following amended paragraph.

[0022] FIG. 1 illustrates the deadlock condition. The process descriptions or blocks in the flow chart presented in FIG. 1 should be understood to represent a somewhat inaccurate overview of an instrumentation process. Those reasonably skilled in the art will understand that an accurate depiction of instrumenting a parent process that executes a vfork to spawn a child process would require a detailed unified modeling language (UML) sequence diagram to reflect the actual interactions in the process. ~~Deadlock condition 10 occurs between development tool 20, parent process 30, and child process 40 as described below. Development tool 20 identifies a process (*e.g.*, parent process 30) that the development tool 20 would like to monitor as indicated in block 22. Next, development tool 20 spawns parent process 30 under trace control as shown in block 24. A process identifier (PID) is assigned to the parent process 30 by the operating system when the parent process is created. Thereafter, in block 26 development tool 20 monitors execution of the parent process by monitoring trace events from parent process 30. Under the UNIX® operating system and its open source and proprietary variants, development tool 20 waits for trace events that include the PID of the parent process. Development tool 20 cannot monitor child process 40, since child process 40 has not been created.~~

Please replace paragraph [0023] (original paragraph [0017]) with the following amended paragraph.

[0023] ~~FIG. 1 illustrates the deadlock condition.~~ Deadlock condition 10 occurs between development tool 20, parent process 30, and child process 40 as described below ~~follows~~. Development tool 20 identifies a process (*e.g.*, ~~instruments~~ parent process 30) that the development tool 20 would like to monitor as indicated in block

22. Next, ~~the~~ development tool 20 spawns parent process 30 under trace control ~~assigns a process identifier (process ID) to the parent process 30~~ as shown in block 24. A process identifier (PID) is assigned to the parent process 30 by the operating system when the parent process is created. Thereafter, in block 26, ~~Next, the~~ development tool 20 monitors execution of the parent process by monitoring trace events using ~~trace control as shown in block 26~~ from parent process 30. Under the UNIX® operating system and its open source and proprietary variants, development tool 20 waits for trace events that include the PID ~~process ID~~ of the parent process ~~as indicated in block 28~~. Development tool 20 cannot monitor child process 40, since child process 40 has not been created.

Please replace paragraph [0024] (original paragraph [0018]) with the following amended paragraph.

[0024] Once parent process 30 is created and started, parent process 30 is instrumented and runs nominally in accordance with its instructions ~~until it encounters a fork instruction (e.g., fork or vfork)~~ as shown in block 32. ~~Thereafter, as shown in block 34, parent process 30 assigns a process ID, different from its own process ID, to identify the child process.~~ The parent process 30 runs until it encounters a vfork system call as shown in block 34. Next, parent process 30 spawns child "A" under trace control as shown in block 36. In accordance with the ~~fork instruction~~ vfork system call, the operating system ~~parent process 30~~ copies ~~itself in its instrumented state to spawn child process 40 and generates a trace event which is received by development tool 20.~~ the current state of the parent process 30 to spawn child process 40 and generates a trace event which is received by development tool 20. Thereafter, as shown in block 38, parent process 30 is essentially suspended waiting for an indication that child process 40 has completed (*e.g.*, indicia of an exec or exit).

Please replace paragraph [0025] (original paragraph [0019]) with the following amended paragraph.

[0025] Once child process 40 is created by the vfork system call ~~fork instruction~~ in parent process 30, and started, child process 40 is instrumented and runs nominally in

accordance with its instructions ~~until it encounters the fork instruction~~ as shown in block 42. ~~Thereafter, as shown in block 44, child process 40 assigns a process ID, different from its own process ID, to identify the subsequent child process. As illustrated in block 46, in accordance with the fork instruction,~~ The child process 40 runs nominally in accordance with its instructions until it encounters the vfork system call shown in block 44. Thereafter, as shown in block 46, child process 40 spawns a child "B" process. At this time, the operating system assigns a PID, different from the parent PID and child "A" PID, to identify the subsequent child process. In accordance with the vfork system call, child process 40 copies itself in its instrumented state to spawn the subsequent child process (not shown) and generates a trace event which is ignored by development tool 20 because development tool 20 is only looking for trace events from parent process 30.

Please replace paragraph [0026] (original paragraph [0020]) with the following amended paragraph.

[0026] Once the vfork system call ~~fork instruction~~ is encountered and processed in child process 40, the deadlock condition has occurred. Parent process 30 is suspended waiting for an indication that child process 40 has completed. Child process 40, which inherited trace control from parent process 30, waits for a process to handle the trace event generated at the time it executed the vfork system call ~~fork instruction~~. Concurrently, development ~~Development~~ tool 20 waits for a trace event from parent process 30.

Please replace paragraph [0027] (original paragraph [0021]) with the following amended paragraph.

[0027] Consequently, it is desirable to have an improved apparatus, program, and method to avoid deadlock induced by a vfork system call ~~for avoiding fork instruction induced deadlocks~~ when monitoring application programming interfaces (APIs) to track the execution of ~~using debugging techniques to instrument and monitor~~ computer programs.

Please replace paragraph [0028] (original paragraph [0022]) with the following amended paragraph.

[0028] An embodiment of a monitoring ~~debug~~ interface, includes a pre-fork event responsive to a vfork system ~~fork instruction~~ call wherein the pre-fork event includes indicia that identifies a child process to be created in accordance with the vfork system ~~fork instruction~~ call.

Please replace paragraph [0029] (original paragraph [0023]) with the following amended paragraph.

[0029] An embodiment of a method for controlling the execution of a child process created from a parent process, where the parent process is monitored ~~instrumented~~ by a software tool includes, receiving indicia that a vfork system call ~~fork instruction~~ will be executed by the parent process, suspending execution of the parent process, extracting a process identifier from the indicia of the vfork system call ~~fork instruction~~, the process identifier corresponding to a child process to be generated by the parent process when the parent process executes the vfork system call ~~fork instruction~~, setting a monitoring thread to observe the child process, and resuming execution of the parent process to enable the parent process to execute past the vfork system call ~~fork instruction~~.

Please replace paragraph [0030] (original paragraph [0024]) with the following amended paragraph.

[0030] Systems and methods for controlling execution of a child process generated by a monitored ~~instrumented~~ parent process are illustrated by way of example and not limited by the implementations in the following drawings. The components in the drawings are not necessarily to scale, emphasis instead is placed upon clearly illustrating the principles used in controlling the execution of such a child process. Moreover, in the drawings, like reference numerals designate corresponding parts throughout the several views.

Please replace paragraph [0032] (original paragraph [0026]) with the following amended paragraph.

[0032] FIG. 2 is a functional block diagram of an embodiment of a computing device.

Please replace paragraph [0039] (original paragraph [0033]) with the following amended paragraph.

[0039] The improved interface provides ~~uses~~ system calls to monitor or otherwise control the execution of threads in monitored processes. A thread is that part of a program that can execute independently of other parts of the program. Operating systems such as UNIX® that support multithreading, enable programmers to design programs whose threaded parts can execute concurrently.

Please delete originally submitted paragraphs [0034] through [0037].

Please replace paragraph [0040] (original paragraph [0038]) with the following amended paragraph.

[0040] ~~In response, the~~ The improved monitoring ~~debug~~ interface includes a pre-fork event and associated processing that can be adapted to multiple tracing facilities such as ttrace and/or ptrace. The improved ~~debug~~ interface and the associated methods described below enable a software tool to control the execution of a child process initiated by an instrumented parent process, where the parent process includes one or more calls to vfork ~~fork instructions~~. While the example embodiments described below are directed to examples where a parent process is instrumented, the present apparatus and methods are applicable to any modified parent process that includes a vfork system call.

Please replace paragraph [0045] (original paragraph [0043]) with the following amended paragraph.

[0045] Software monitor 300 and application(s) 224 include one or more source programs, executable programs (object code), scripts, or other collections each comprising a set of instructions to be performed. As will be explained in detail below, software monitor 300 includes logic that controls the execution of application(s) 224. More specifically, software monitor 300 includes logic that controls the execution of a child process or thread generated by a modified ~~instrumented~~ parent process found within application(s) 224 where the parent process or thread includes a vfork system call ~~fork instruction~~. As illustrated in FIG. 2, software monitor 300 includes an instrumentation engine 310, a process monitor 320, and a monitoring interface 330. It should be well-understood by one skilled in the art, after having become familiar with the teachings of the improved ~~debug~~ interface, that software monitor 300 and application(s) 224 may be written in a number of programming languages now known or later developed. Moreover, software monitor 300 and application(s) 224 may be stored across distributed memory elements in contrast with the memory 220 shown in FIG. 2.

Please replace paragraph [0050] (original paragraph [0048]) with the following amended paragraph.

[0050] Those skilled in the art will understand that various portions of software monitor 300 can be implemented in hardware, software, firmware, or combinations thereof. In a preferred embodiment, software monitor 300 is implemented using software that is stored in memory 220 and executed by a suitable instruction execution system. If implemented solely in hardware, as in an alternative embodiment, software monitor 300 can be implemented with any or a combination of technologies well-known in the art (*e.g.*, discrete logic circuits, application-specific integrated circuits (ASICs), programmable-gate arrays (PGAs), field-programmable gate arrays (FPGAs), *etc.*), or technologies later developed.

Please replace paragraph [0051] (original paragraph [0049]) with the following amended paragraph.

[0051] In one ~~a preferred~~ embodiment, the software monitor 300 is implemented via a combination of software and data stored in memory 220 and executed and stored or otherwise processed under the control of processor 210. It should be noted, however, that software monitor 300 is not dependent upon the nature of the underlying processor 210 or memory 220 in order to accomplish designated functions.

Please replace paragraph [0052] (original paragraph [0050]) with the following amended paragraph.

[0052] Reference is now directed to FIG. 3, which presents a functional block diagram of an embodiment of software monitor 300. In the present embodiment, software monitor 300 instruments or otherwise modifies code. In other embodiments, software monitor 300 may employ sampling techniques that do not necessitate modification of a parent process. As illustrated in FIG. 3, software monitor 300 comprises the instrumentation engine 310, process monitor 320, and a monitoring ~~debug~~ interface 330. Before software monitor 300 can collect and analyze performance information regarding a specific thread or process, instrumentation engine 310 inserts or otherwise modifies code in ~~into~~ the target process or thread. Preferably, software monitor 300 contains logic that in accordance with dynamic binary-instrumentation techniques, instruments or modifies only those portions of parent process 350 that will be executed by processor 210.

Please replace paragraph [0053] (original paragraph [0051]) with the following amended paragraph.

[0053] Instrumentation engine 310 may receive data via various input/output devices 230, data stored in memory 220 (FIG. 2), as well as various application(s) 224. The data will identify one or more target processes or threads to instrument. In addition, the data may include various parameters and flags that instrumentation engine 310 uses in generating parent process 350. Alternatively, instrumentation engine 310 can be

programmed with one or more default parameters to apply when modifying ~~instrumenting~~ the target process. Instrumentation engine 310, having received data identifying the target process or thread applies the various parameters and flags and generates parent process 350. Parent process 350 is an instrumented or modified version of the identified target process or thread.

Please replace paragraph [0054] (original paragraph [0052]) with the following amended paragraph.

[0054] As further illustrated in the functional block diagram of FIG. 3, software monitor 300 includes process monitor 320. Process monitor 320 includes logic for coordinating the collection of data during execution of parent process 350. When parent process 350 includes one or more vfork system calls ~~fork instructions~~, process monitor 320 ensures that data collected during execution of both the parent process 350 and its child process 352 are associated with the process responsible for generating the data. As shown in FIG. 3, process monitor 320 functions through monitoring ~~debug~~ interface 330 and the underlying operating system 222. Information flows between process monitor 320 and monitoring ~~debug~~ interface 330 can include monitored system calls, events, and signals 332.

Please replace paragraph [0055] (original paragraph [0053]) with the following amended paragraph.

[0055] Monitoring ~~debug~~ interface 330 includes logic for receiving and responding to the various trace system calls, events, and signals 332. In addition, monitoring ~~debug~~ interface 330 includes instruction generator 335. Instructions 336 are in accordance with the underlying operating system 222. As illustrated in FIG. 3, monitoring ~~debug~~ interface 330 receives and responds to trace system calls, events, and signals 332 generated and sent by parent process 350, child process 352, and process monitor 320.

Please replace paragraph [0056] (original paragraph [0054]) with the following amended paragraph.

[0056] Operating system 222 as illustrated in FIG. 3, sends and receives instructions 336 ~~335~~ both to and from monitoring ~~debug~~ interface 330 via instruction interface 338. In addition, operating system 222 receives a vfork system call ~~fork instruction 334 (e.g., fork or vfork)~~ via connection 334 from parent process 350. As provided ~~described~~ in the UNIX® operating system and many of its proprietary and open source derivatives, a vfork system call ~~fork instruction 334~~ suspends execution of parent process 350 and generates child process 352 which contains a copy of the instrumented code and trace calls, events, and signals contained within parent process 350.

Please replace paragraph [0057] (original paragraph [0055]) with the following amended paragraph.

[0057] However, in addition to the other trace system calls, events, and signals 332, parent process 350 communicates a pre-fork event 375 to monitoring ~~debug~~ interface 330. Pre-fork event 375 includes indicia identifying child process 352 before it is created by a subsequently executed vfork system call ~~fork instruction~~ within parent process 350. The indicia includes at least a process identifier of the child process 352. Monitoring ~~debug~~ interface 330 further includes logic configured to recognize and respond to the pre-fork event 375.

Please replace paragraph [0058] (original paragraph [0056]) with the following amended paragraph.

[0058] While the functional block diagram presented in FIG. 3 illustrates software monitor 300 as having a single centrally-located instrumentation engine 310 with co-located process monitor 320 and monitoring ~~debug~~ interface 330, it should be understood that the various functional elements of software monitor 300 may be distributed across multiple locations in memory 220 and/or across multiple memory devices (not shown). It should be further understood that instrumentation engine 310 is not limited to dynamic binary instrumentation techniques and may include logic in

accordance with binary instrumentation techniques (*i.e.*, logic that instruments all portions of the identified parent process 350) and statistical sampling and other applications that can exploit the pre-fork event 375.

Please replace paragraph [0059] (original paragraph [0057]) with the following amended paragraph.

[0059] Reference is now directed to the flow chart illustrated in FIG. 4. In this regard, the various functions shown in the flow chart present a method for controlling the execution of a child process created from a parent process, where the parent process is instrumented by a software tool that may be realized by software monitor 300. As illustrated in FIG. 4, the method may begin by determining whether it is desired to instrument a parent process as shown in query 402. When it is determined that it is desirable to instrument the parent process as indicated by the flow control arrow labeled "YES" that exits query 402, the method responds by acquiring desired monitoring parameters, as indicated in block 404. Otherwise, as indicated by the flow control arrow labeled "NO_x" the method bypasses blocks 404 through 408 and waits for an indication of a pre-fork event ~~for controlling the execution of a child process created from a parent process can be bypassed~~.

Please replace paragraph [0060] (original paragraph [0058]) with the following amended paragraph.

[0060] Returning to the condition where it is desired to instrument the parent process, as illustrated in block 406, a process monitor thread is set to communicate with the parent process. As shown in block 408, the parent process is executed and instrumented ~~and executed~~ to extract or otherwise record desired parameters.

Please replace paragraph [0061] (original paragraph [0059]) with the following amended paragraph.

[0061] Thereafter, as indicated in the wait loop formed by query 410 and wait block 424 ~~412~~, execution of the parent process continues until a pre-fork event is detected. When a pre-fork event is detected, as indicated by the flow control arrow labeled "YES" exiting

query 410, the function indicated in block 412 ~~414~~ is performed. More specifically, the process identifier of the child process that will be created by the subsequent vfork system call ~~fork instruction~~ is extracted from the pre-fork event. Thereafter, as indicated in block 414 ~~416~~, a process monitor thread is configured to respond to trace events generated by the future child process. Next, as shown in block 416 ~~418~~, execution of the child ~~parent~~ process resumes ~~to permit the parent process to execute the subsequent fork instruction~~.

Please replace paragraph [0062] (original paragraph [0060]) with the following amended paragraph.

[0062] As described above, a child process that contains a copy of the parent process (including code inserted by an instrumentation process) is generated and executed in accordance with the vfork system call ~~fork instruction~~. In addition, the parent process is suspended until the child process calls an exec or an _exit system call ~~terminates (e.g., the child process generates an exec or exit event)~~. When it is determined that the child process has invoked one of those calls or has been terminated by the operating system, as indicated by the flow control arrow labeled "YES" exiting query 418 ~~420~~, the process monitor is configured to monitor trace events generated by the parent process, as indicated in block 422 ~~424~~, before resuming execution of the parent process.

Please replace paragraph [0063] (original paragraph [0061]) with the following amended paragraph.

[0063] Execution of the parent process continues, as indicated in the wait loop formed by query 426 and wait block 424 ~~428~~, until the parent process generates an exec or _exit call or is terminated by the operating system ~~terminates (e.g., the parent process generates an exec or exit event)~~. When it is determined that the parent process has generated one of the exec or _exit calls or has been terminated, as indicated by the flow control arrow labeled "YES" exiting query 426, the process monitor is suspended as indicated in block 428 ~~430~~ and run-time data observed during execution of the parent process is collected, analyzed, and presented as shown in block 430 ~~432~~.

Please replace paragraph [0064] (original paragraph [0062]) with the following amended paragraph.

[0064] Those skilled in the art will understand that the method for controlling the execution of a child process created from a parent process illustrated in FIG. 4 ~~is configured to successfully control a single child process generated as the result of the execution of the first fork instruction within an instrumented parent process~~ can be extended to control any number of processes, for example by using separate threads to monitor each process of interest. Since a given process can only execute a single vfork system call at a time (since it becomes blocked while executing the system call), the design therefore addresses the general case of multiple monitored processes that may call vfork. A software monitor may include multiple threads for controlling the performance of the desired functions. For example,a first thread may continuously wait fro and process pre-fork events. A parallel (i.e., simultaneously executed) thread may continuously wait for an indication that the presently active process has terminated. These parallel threads may be executed as desired by a software monitor.

Please replace paragraph [0065] (original paragraph [0063]) with the following amended paragraph.

[0065] FIG. 5 is a flow chart illustrating an embodiment of a method for executing a parent process instrumented by a software tool to ensure execution of a child process when the parent process contains a vfork system call ~~fork instruction~~. As illustrated in query 502, the parent process 350 begins by determining if a vfork system call ~~fork or vfork instruction~~ is about to be executed by the parent process or thread. When it is determined that a vfork system call ~~fork or vfork instruction~~ is about to be executed by the parent process or thread as indicated by the flow control arrow labeled "YES" that exits query 502, the parent process generates a pre-fork event as indicated in block 504. Next, as shown in block 506, the parent process sends the pre-fork event to the software tool responsible for monitoring ~~instrumenting~~ the parent process.

Please replace paragraph [0066] (original paragraph [0064]) with the following amended paragraph.

[0066] Thereafter, as indicated in the wait loop formed by query 508 and wait block 510, execution of the parent process is suspended until after the parent receives an indication from the software tool that the pre-fork event has been successfully processed. When the pre-fork event has been processed, as indicated by the flow control arrow labeled "YES" exiting query 508, the parent is activated and executes the vfork system call ~~fork instruction~~ as shown in block 512. Once the vfork system call ~~fork instruction~~ has been executed, the parent is suspended as indicated in block 514.

Please replace paragraph [0067] (original paragraph [0065]) with the following amended paragraph.

[0067] As indicated in the wait loop formed by query 516 and wait block 518, the parent process remains suspended until the parent receives an indication that the child process has invoked an exec or an _exit system call, thus generating corresponding events. When the child process has invoked the exec or _exit system call ~~terminated~~, as indicated by the flow control arrow labeled "YES" exiting query 516, the parent process resumes as shown in block 520. As indicated in query 522, the parent process continues until the process generates an exit event ~~it terminates nominally and sends an exec event or fails and sends an exit event~~. As shown by the flow control arrow labeled "NO" exiting query 522, the parent is configured to report any future vfork system call ~~fork or vfork instructions~~ by repeating the functions and queries described above.

Please replace paragraph [0068] (original paragraph [0066]) with the following amended paragraph.

[0068] Those skilled in the art will understand that the method for executing a parent process monitored ~~instrumented~~ by a software tool to ensure execution of a child process when the parent process contains a vfork system call ~~fork instruction~~ illustrated in FIG. 5 is configured to generate and send a pre-fork event before executing a vfork system call ~~fork or vfork instruction~~. The parent process or thread

may be implemented via multiple threads for controlling the performance of the desired functions. A given thread may be responsible for handling monitoring events generated by a single process or from multiple processes. Multiple such ~~For example, a first thread may continuously identify when a fork instruction is encountered in the instruction sequence. A second thread may intermittently be started to wait for an indication that the software tool has successfully processed the pre-fork event. A third thread may be responsible for handling trace events generated by the child process. These and other~~ threads may be executed as may be desired by a parent process or thread to implement the various functions illustrated in FIG. 5.

Please replace paragraph [0069] (original paragraph [0067]) with the following amended paragraph.

[0069] Reference is now directed to the flow chart illustrated in FIG. 6, which illustrates an embodiment of a method for controllably switching a target process of a process monitor thread between an instrumented parent process and a child process generated by the parent process. In this regard, process monitor 320 begins with query 602 where it is determined if a target PID has been received ~~the child process has been successfully generated and started~~. If the result of query 602 indicates that the PID has not been received ~~child process has not started successfully, process moniotr 320 is configured to wait as indicated in block 604~~, as indicated by the flow control arrow labeled "NO" exiting query 602, processing continues after wait block 604. Once query 602 indicated that a target PID has been received, ~~Next,~~ query 606 is performed to determine if the child process has started ~~parent process has received an indication that the fork instruction failed. When the parent process receives an indication that the fork instruction failed as indicated by the flow control arrow labeled "YES,"~~ . If the result of query 606 indicates that the child process has not started successfully, process monitor 320 is configured to determine if the vfork system call issued by the parent process has failed as indicated in query 608. When the parent process vfork system call has not failed, query 606 is repeated after wait block 610. When the result of query 608 indicates that the parent vfork system call has failed, as indicated by the flow control arrow labeled "YES" exiting query 608, the process monitor 320 is configured to notify the software monitor that the vfork

system call failed as indicated in block 612 ~~sets the active process identifier (PID) to the parent process' PID as shown in block 608. Otherwise, the process monitor returns to query 602. The determinations made in query 602 and query 604 are repeated to handle the case where an event documenting the creation of the child process as a result of the fork instruction has not been received before the process monitor is started~~.

Please replace original paragraphs [0068] and [0069] with the following paragraph.

[0070] Otherwise, when query 606 indicates that the child process has successfully started as indicated by the flow control arrow labeled "YES" exiting query 606, query 614 is performed to determine if the parent vfork system call failed. When query 614 indicates that the parent vfork system call has not failed, block 616 is bypassed. Otherwise, when query 614 indicates that the parent vfork system call has failed, process monitor 320 sets a target PID to the child process' PID as indicated in block 616. Thereafter, process monitor 320 monitors trace events from the target PID as indicated in block 618. Process monitor 320 continues to monitor trace events from the child process associated with target PID until the process executes an exec or _exit system call or is otherwise terminated by the operating system.

Please replace paragraph [0071] (original paragraph [0070]) with the following amended paragraph.

[0071] FIG. 7 is a flow chart of an embodiment of a method 700 for controlling the execution of a child process. When an appropriately configured software tool receives indicia that a vfork system call ~~fork instruction~~ is about to be executed by a parent process, as indicated in block 702, the software tool suspends execution of the parent process as illustrated in block 704. While the parent process is suspended, as indicated in block 706, the software tool extracts a process identifier from the indicia of the vfork system call ~~fork instruction~~ corresponding to a child process to be generated by the process when the parent executes the pending vfork system call ~~fork instruction~~. In block 708, the software tool sets a process monitor thread to observe the execution of

the child process. The software tool waits until the child process calls exec or _exit or the child process is terminated by the operating system as shown in block 710. Once, the software tool receives an indication that the child process has ended, the ~~software~~ tool ~~then~~ resumes execution of the parent process as indicated in block 712 ~~710~~.

Please replace paragraph [0072] (original paragraph [0071]) with the following amended paragraph.

[0072] FIG. 8 is a flow chart of an embodiment of a method for executing an instrumented parent process to ensure execution of a child process. When an appropriately configured software tool receives indicia that a vfork system call ~~fork instruction~~ is about to be executed by a parent process, as indicated in block 802, the software tool generates a pre-fork event that includes indicia of a child process that will be generated by the vfork system call ~~fork instruction~~, as shown in block 806. Otherwise, the software tool waits for an amount of time to pass, as illustrated by the flow control arrow labeled "NO" exiting block 802 and wait block 804 before repeating the query of decision block 802.

Please replace paragraph [0074] (original paragraph [0073]) with the following amended paragraph.

[0074] Otherwise, when it is determined that the pre-fork event has successfully terminated, as indicated by the flow control arrow labeled "YES" ~~"Yes"~~ exiting decision block 810, the software tool suspends execution of the parent process ~~executes the fork instruction~~ as shown in block 814. Thereafter, the software tool directs the creation of execution of the child process as shown in block 816 ~~execution of the parent process is suspended~~ to enable the software tool to monitor execution of the child process.

Please replace paragraph [0075] (original paragraph [0074]) with the following amended paragraph.

[0075] Any process descriptions or blocks in the flow charts presented in FIGs. 4-8 should be understood to represent modules, segments, or portions of code or logic, which include one or more executable instructions for implementing specific logical

functions in the associated process. Alternate implementations are included within the scope of the disclosed methods in which functions may be executed out-of-order from that shown or discussed, including substantially concurrently or in reverse order, depending on the functionality involved, as would be understood by those reasonably skilled in the art after having become familiar with the improved monitoring ~~debug~~ interface and the associated methods for controlling the execution of a child process generated by an instrumented parent process.

Please replace the abstract with the following amended abstract.

[0076] A monitoring ~~debug~~ interface, includes logic responsive to a pre-fork event, the pre-fork event responsive to a vfork system ~~fork instruction~~ call ~~wherein the~~. The pre-fork event includes indicia that identifies a child process to be created in accordance with the vfork system ~~fork instruction~~ call. A method for controlling the execution of a child process created from a parent process, where the parent process is monitored ~~instrumented~~ by a software tool includes, receiving indicia that a vfork system call ~~fork instruction~~ will be executed by the parent process, suspending execution of the parent process, extracting a process identifier from the indicia of the vfork system call ~~fork instruction~~, the process identifier corresponding to a child process to be generated by the parent process when the parent process executes the vfork system call ~~fork instruction~~, setting a process monitor thread to observe the child process, and resuming execution of the parent process to enable the parent process to execute past the vfork system call ~~fork instruction~~.